Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 3: this Friday @ 11:59 pm
  - Lab 2: next Monday @ 11:59 pm
  - Assignment 1: Monday Oct 10th @ 11:59 pm
- Please include all instructors when sending private messages on Piazza, if possible
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous lecture

- Red-Black BST (self-balancing BST)
  - definition and basic operations
  - delete
  - runtime of operations
- Turning recursive tree traversals to iterative

Muddiest Points

- **Q: So, just to recap: When we add a new node to a Red-Black BST, if that leads to a violation we need to fix that violation and all others back up to the tree's root before we can say we're done?**
- Yep!

Muddiest Points

- **Q: I think there may be an overlap in which sections muddiest points you are displaying**
- Yes. But I made sure to explain the concept before going over the muddies points

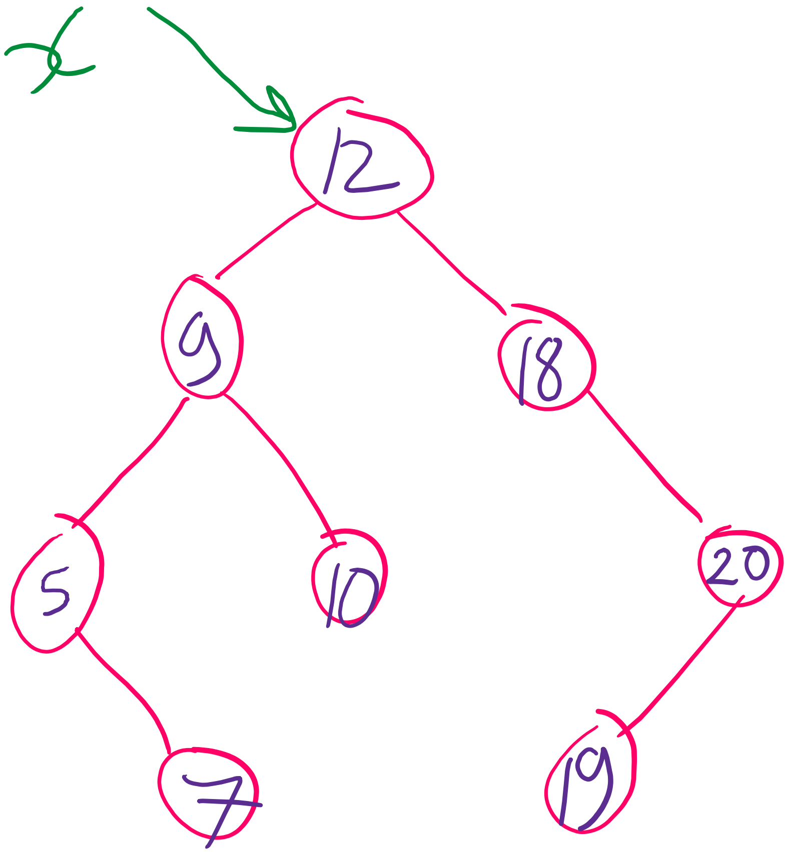
Muddiest Points

- **Q: how does a tree map work? a hashmap works effectively by 'converting' a string to an int by using a hash code to map a key to a value, what is a treemap and how does it do that?**
- Both TreeMap and HashMap implement the Map or Dictionary interface
- Nothing in the Map interface requires the conversion into an integer
- TreeMap uses a Binary Search Tree instead of a hash table to perform add, search, delete, etc.

Muddiest Points

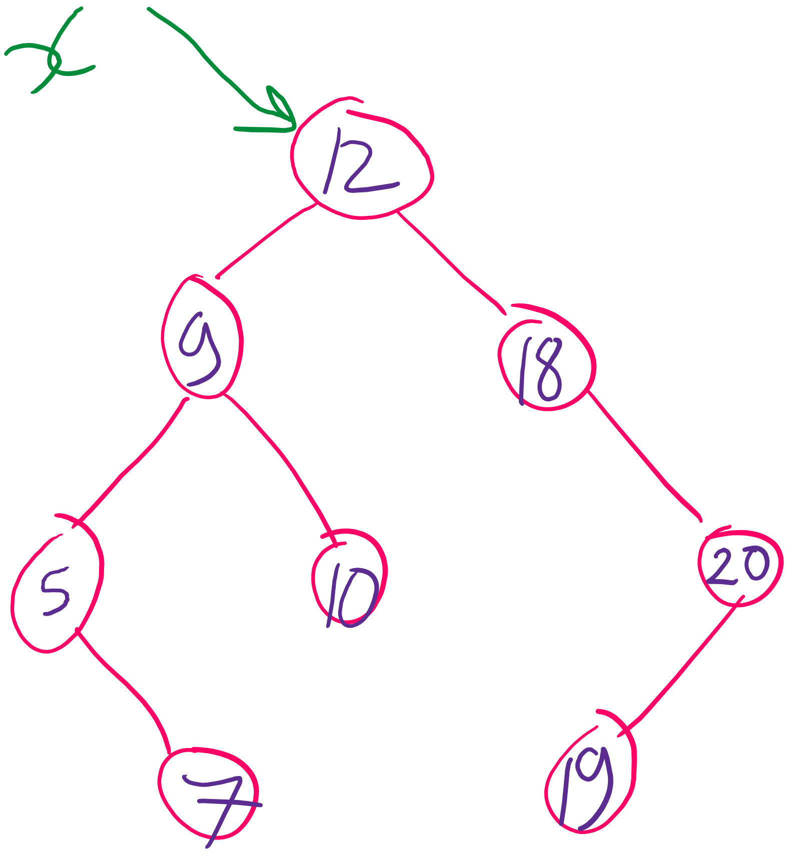
- **Q: Could you please explain the iterative approach to the BST is again?**
- Sure.

BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if <  x = x.left
    if >  x = x.right
}
```

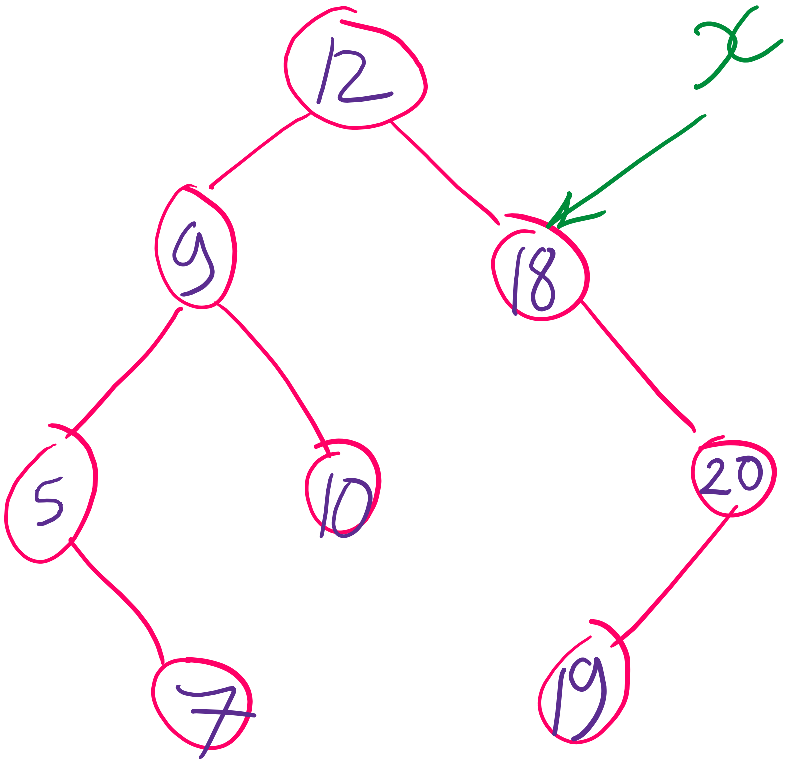
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if <  x = x.left
    if >  x = x.right
}
```

Search for 19

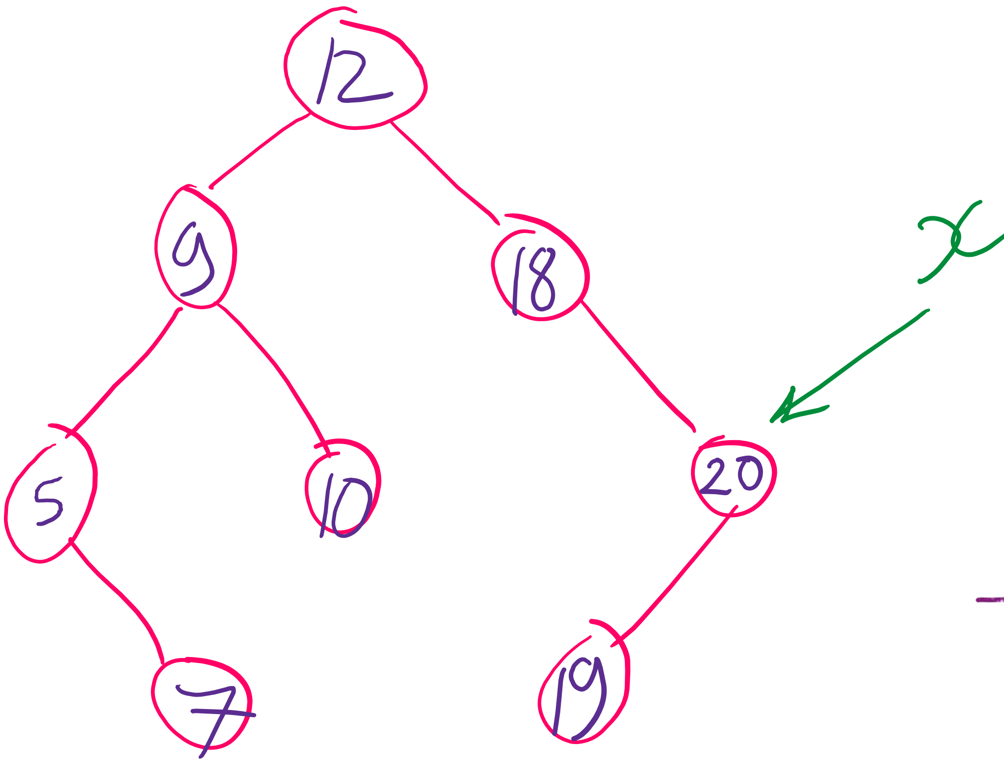
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if <  x = x.left
    if >  x = x.right
}
```

Search for 19

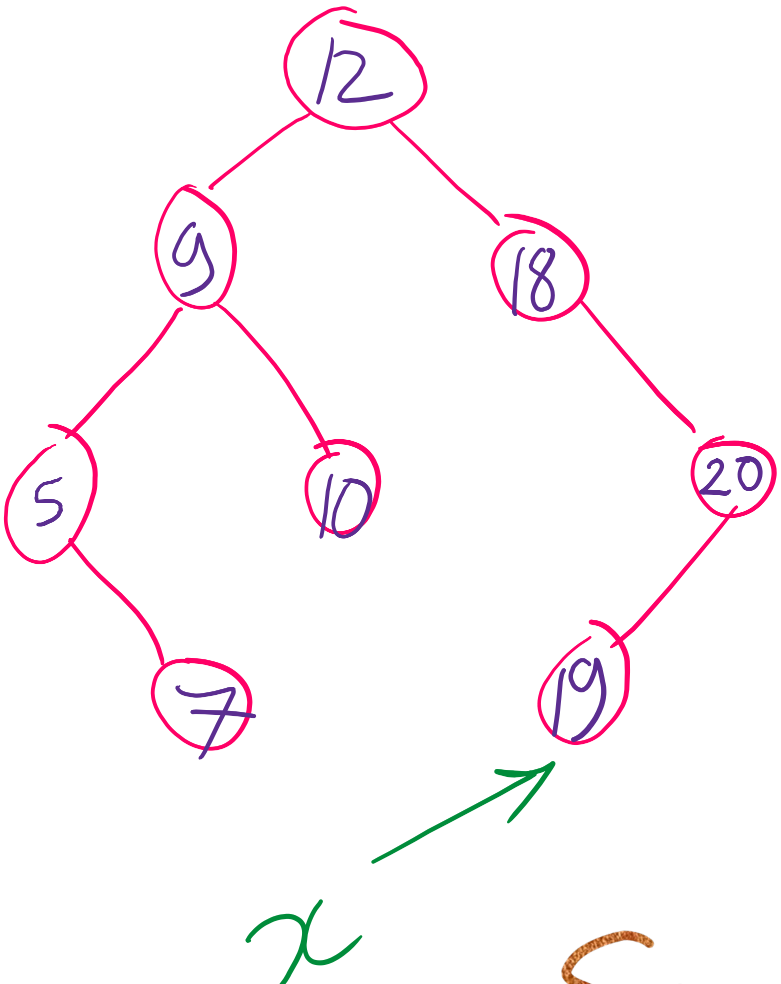
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if < x = x.left
    if > x = x.right
}
```

Search for 19

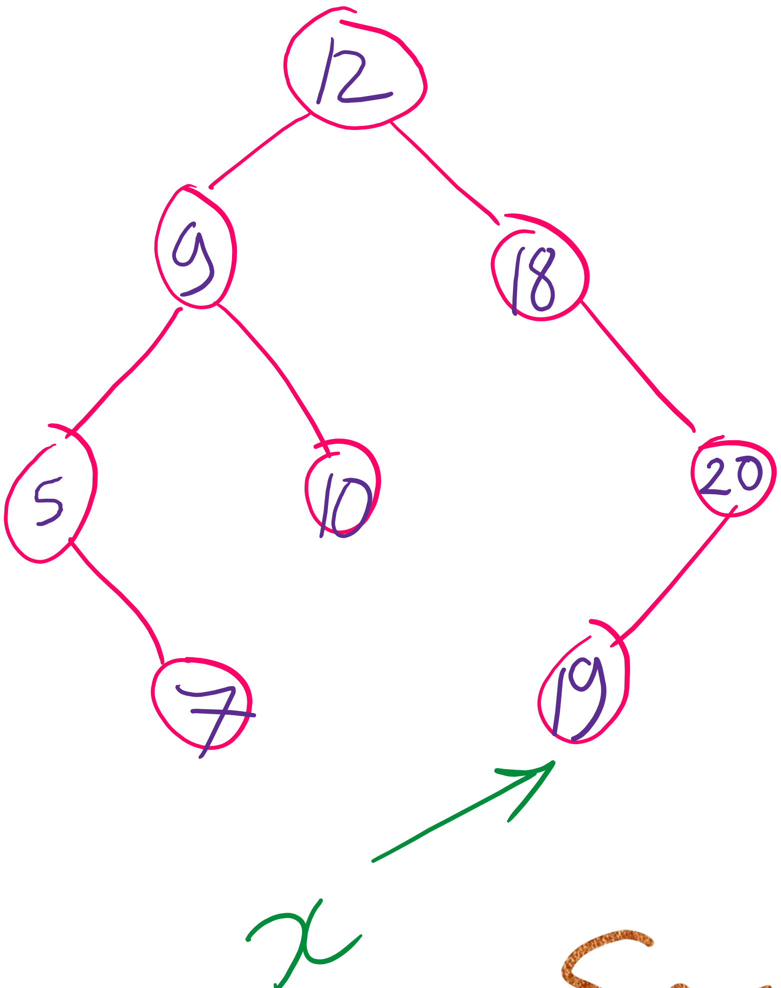
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if <  x=x.left
    if >  x=x.right
}
```

Search for 19

BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if <  x=x.left
    if >  x=x.right
}
```

Search for 19

Muddiest Points

- **Q: The iterative method for pre-order was confusing for me to grasp but it started to make more sense once we did in-order and post-order.**
- Thanks!

Muddiest Points

- **Q: How does a Red Black BST keep track of the data in the nodes if it's just comparing colors?**
- It is comparing data as we go down the tree (except for delete) and checking colors as we climb back up the tree

Muddiest Points

- **Q: can you explain the stack and what pushing/popping are?**
- Stack is an abstract data type in which data items are ordered in a Last In First Out order.
- Pushing into a Stack means to add an item at the “top” of the stack
- Popping means to remove the top item

This Lecture

- Binary Search Tree uses comparisons between keys to guide the searching
- What if we use the digital representation of keys for searching instead?
    - Keys are represented as a sequence of digits (e.g., bits) or alphabetic characters
- Digital Searching Problem

Digital Searching Problem

- Input:
  - a (large) dynamic set of data items in the form of
    - *n* (key, value) pairs; key is a string from an alphabet of size *R*
    - Each key has *b* bits or *w* characters (the chars are from the alphabet)
    - What is the relationship between *b* and *w*?
  - a *target key (k)*
- Output:
  - The corresponding value to *k* if target key found
  - Key not found otherwise

Digital Search Trees (DSTs)

Instead of looking at less than/greater than, lets go left or right based on the bits of the key

So, we again have 4 options:

- current node is null, k not found
- k is equal to the current node's key, k is found, return corresponding value
- current bit of k is 0, continue to left child
- current bit of k is 1, continue to right child

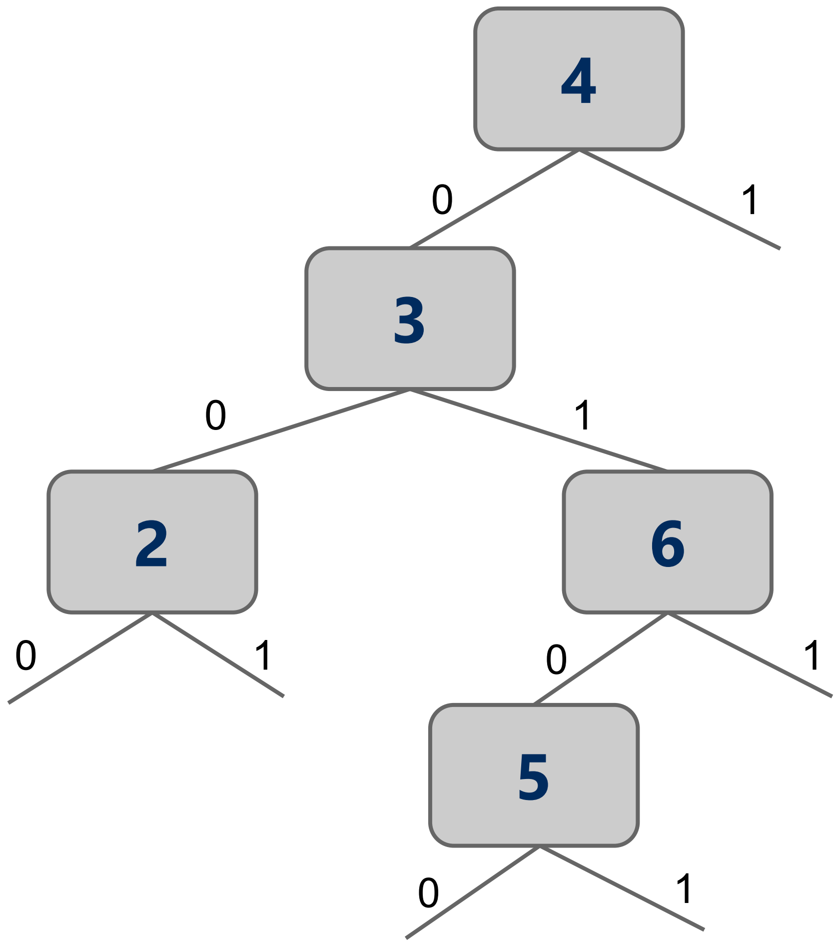
DST example: Insert and Search

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

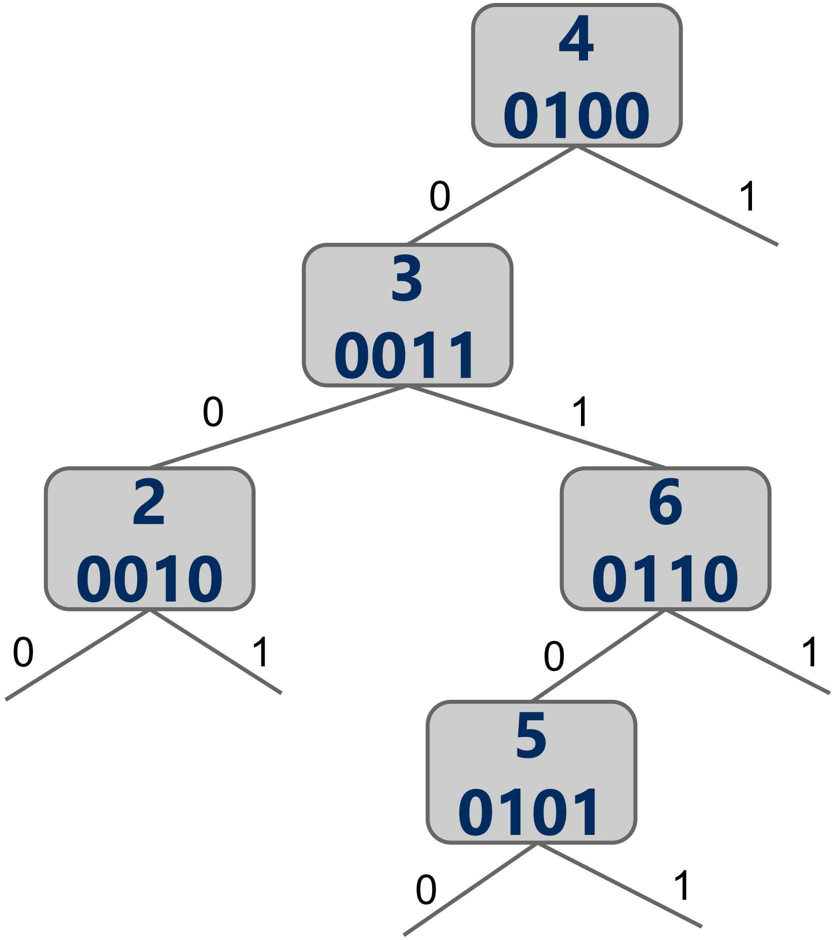
Search:

3	0011
7	0111



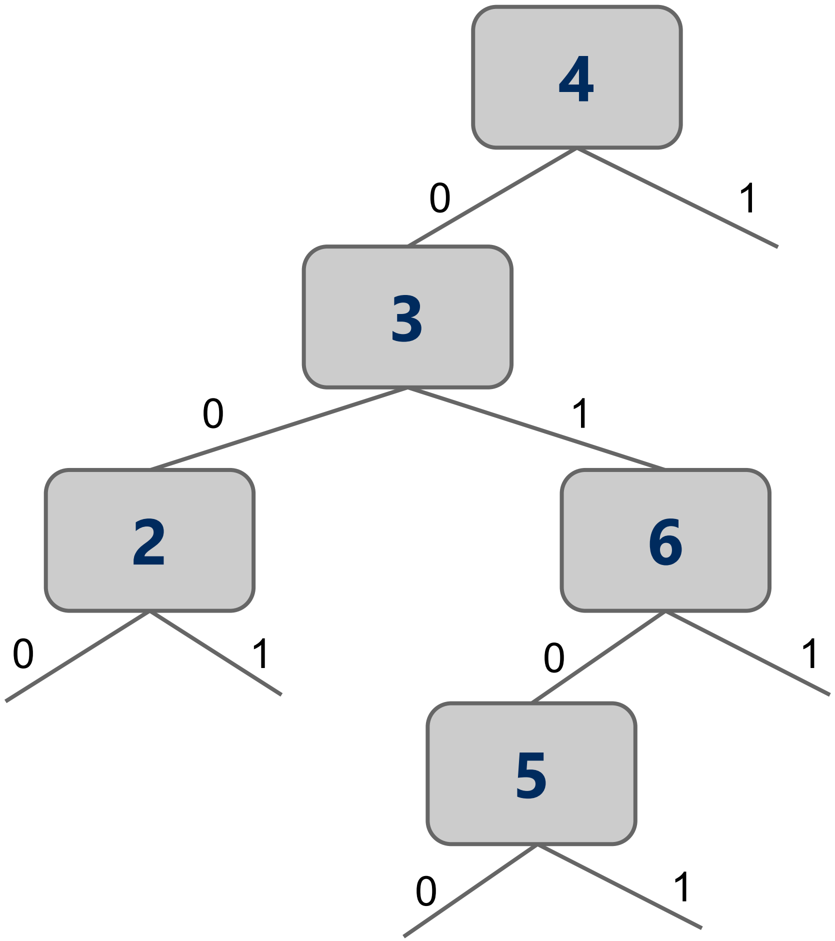
DST and Prefixes

- In a DST, each node shares a **common prefix** with all nodes in its subtree
  - E.g., 6 shares the prefix "01" with 5
- In-order traversal doesn't produce a sorted order of the items
  - Insertion algorithm can be modified to make a DST a BST at the same time



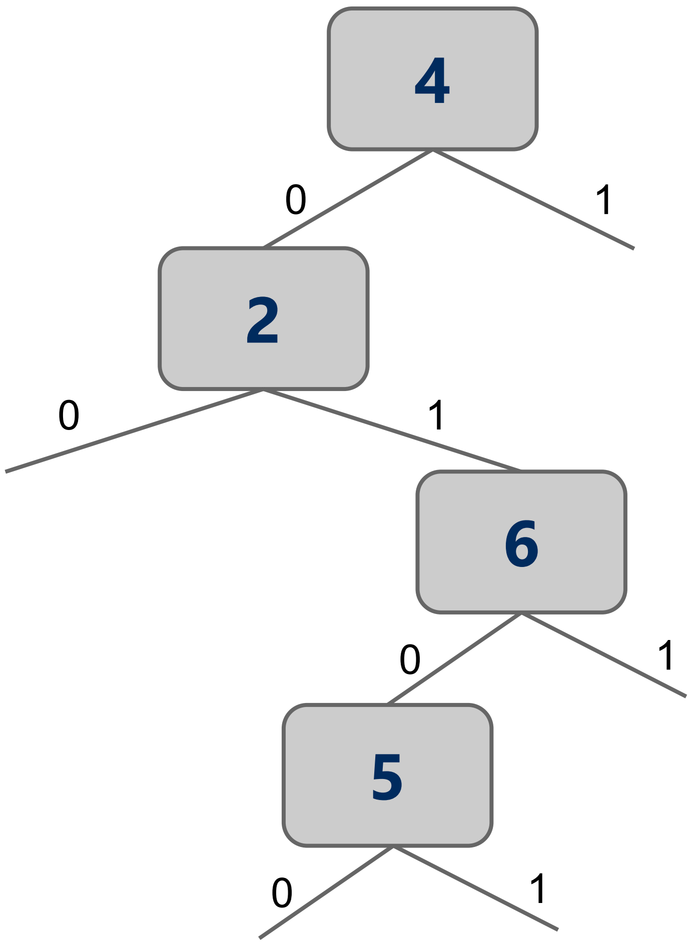
DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5



DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5

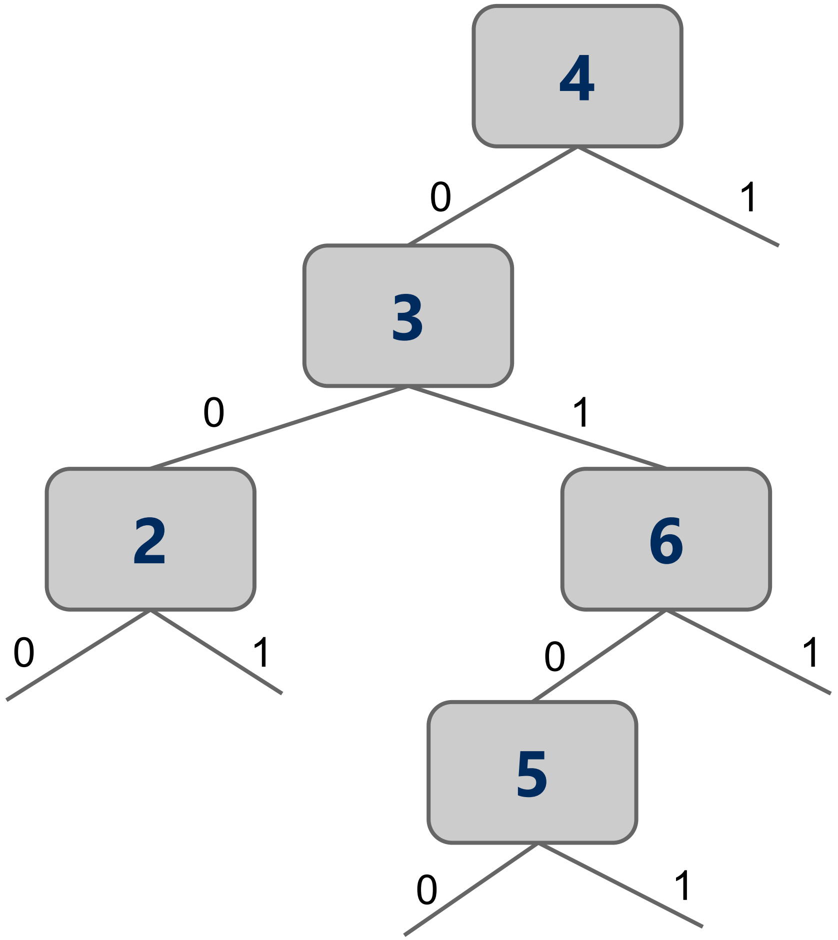


DST example: Variable length keys

- Insert

1 01

- Must be in place of 6
- Replace 6 by 1 and re-insert 6



DST example: Variable length keys

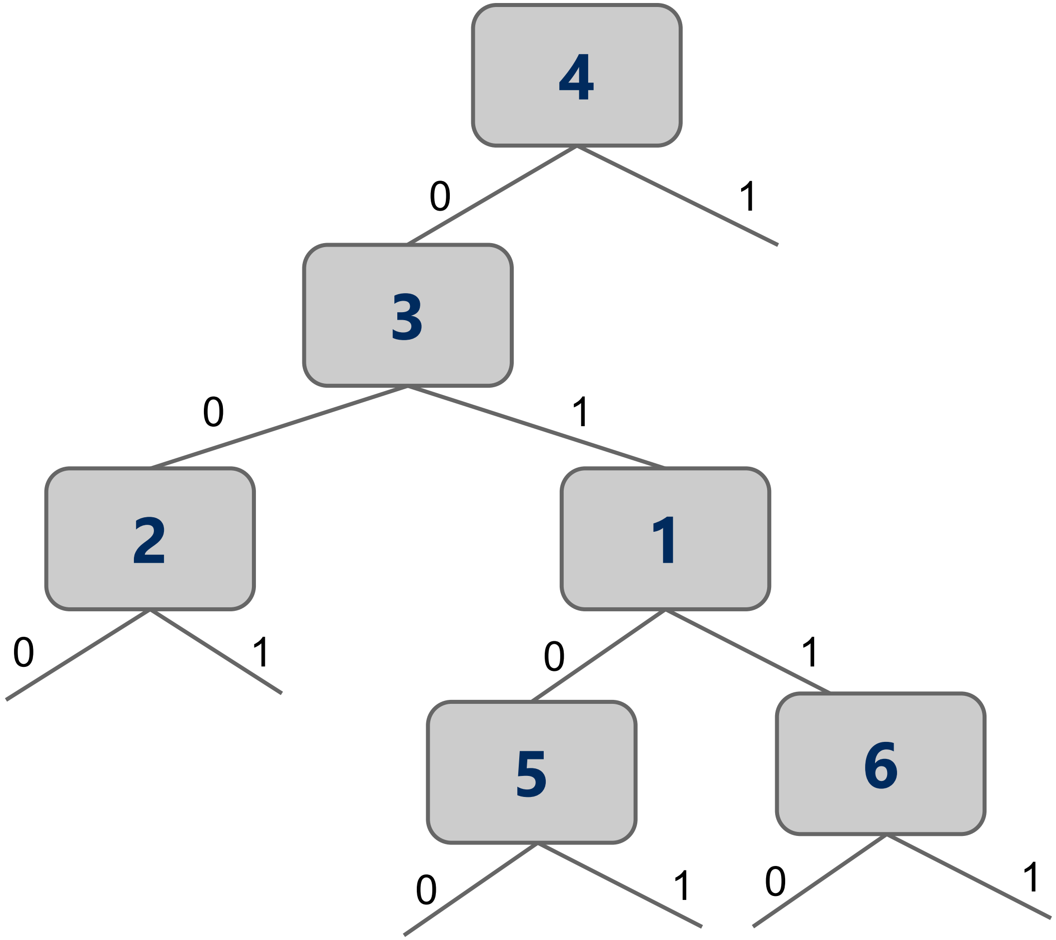
- Insert

1   01

- Must be in place of 6

- Replace 6 by 1 and re-insert

6   0110



Analysis of digital search trees

$$\text{average case runtime} = \sum_{\text{all cases}} Pr(\text{Case}_i) \times \text{runtime for Case}_i$$

- Runtime?
  - O(b), b is the bit length of the target or inserted key
  - On average, b = log(n)
  - When branching according to a 0 or 1 is equally likely
  - In general b >= $\lceil \log n \rceil$
- We end up doing many **equality** comparisons against the full key
- This is better than less than/greater than comparison in BST
- Can we improve on this?

Radix search tries (RSTs)

- Trie as in retrieve, pronounced the same as "try"
- Instead of storing keys inside nodes in the tree, we store them implicitly as paths down the tree
  - Interior nodes of the tree only serve to direct us according to the bitstring of the key
  - Values can then be stored at the end of key's bitstring path (i.e., at leaves)
  - RST uses less space than BST and DST

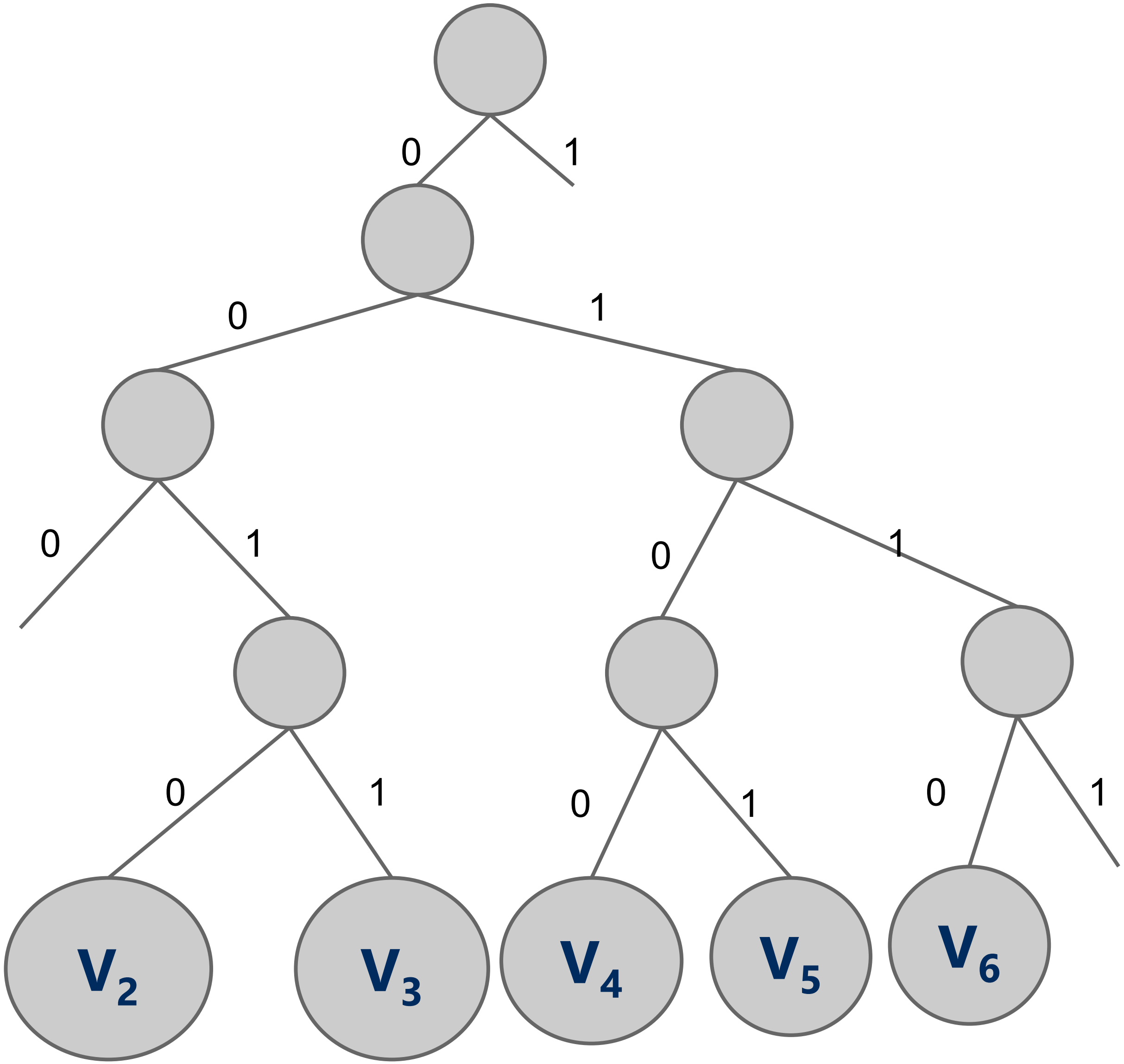
RST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

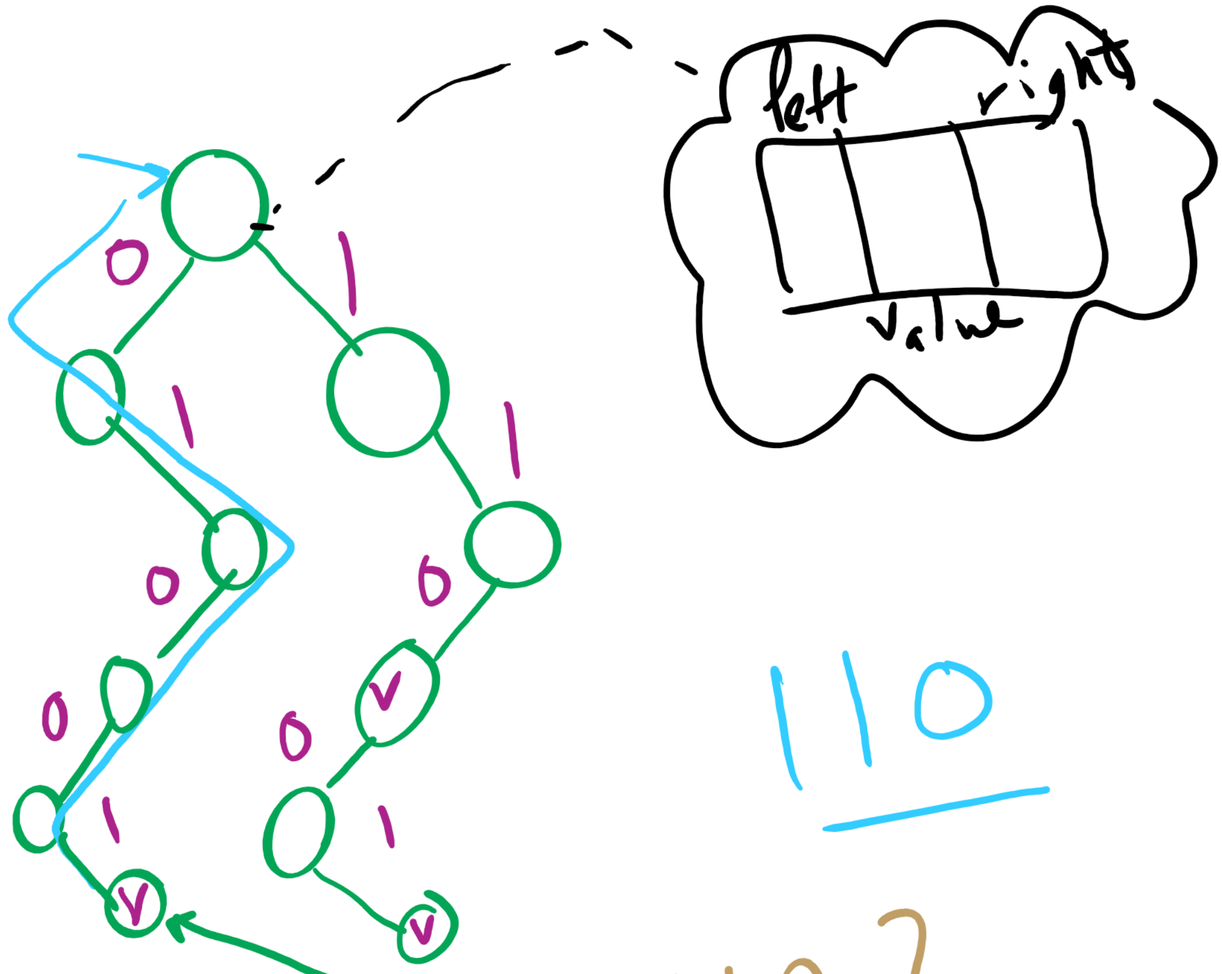
Search:

3	0011
7	0111



Binary RST

11001



110

$\boxed{01001}$

010?

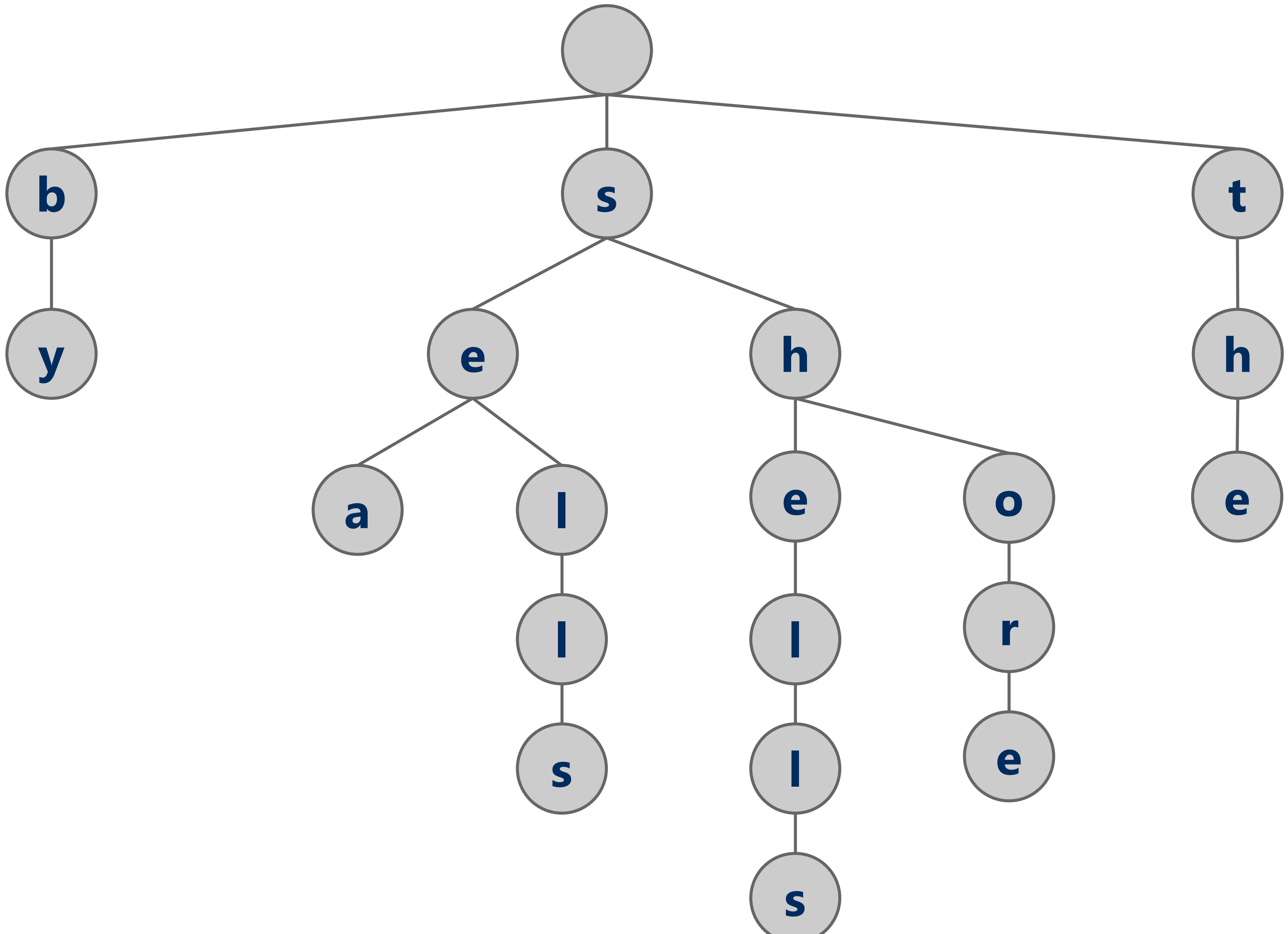
RST analysis

- Runtime?
- O(b), the bit length of the key
    - However, this time we don't have full key comparisons
- Would this structure work as well for other key data types?
- Characters?
    - Characters are the same as 8-bit ints (assuming simple ascii)
- Strings?
- May have huge bit lengths
- How to store Strings?

Larger branching factor tries

- In our binary-based Radix search trie, we considered one bit at a time
- What if we applied the same method to characters instead of bits in a string?
  - What would this new structure look like?
  - How many children per node?
    - up to R (the alphabet size)
  - Also called R-way radix search tries
- Let's try inserting the following strings into an trie:
  - she, sells, sea, shells, by, the, sea, shore

Another trie example



Analysis

- Runtime?
- $\Theta(w)$ where w is the character length of the string
  - So what do we really gain over RSTs?
    - For strings, w < b, and overall tree height is reduced
    - $w = \dfrac{b}{\log R}$ where R is the alphabet size
    - For binary RST: average tree height = $\log_2(n)$
    - For R-way RST: average tree height = $\log_R(n)$

Further analysis

- Search Miss
  - Require an average of $\log_R(n)$ nodes to be examined
    - Where R is the size of the alphabet being considered
    - Proof in Proposition H of Section 5.2 of the text
  - Average # of checks with 2^{20} keys in an RST?
    - $\log_2 n = \log_2 2^{20} = 20$
  - With 2^{20} keys in a large branching factor trie, assuming 8-bits at a time?
    - $\log_R n = \log_{256} 2^{20} = \log_{256} (2^8)^{2.5} = = \log_{256} 256^{2.5} = 2.5$

Implementation Concerns

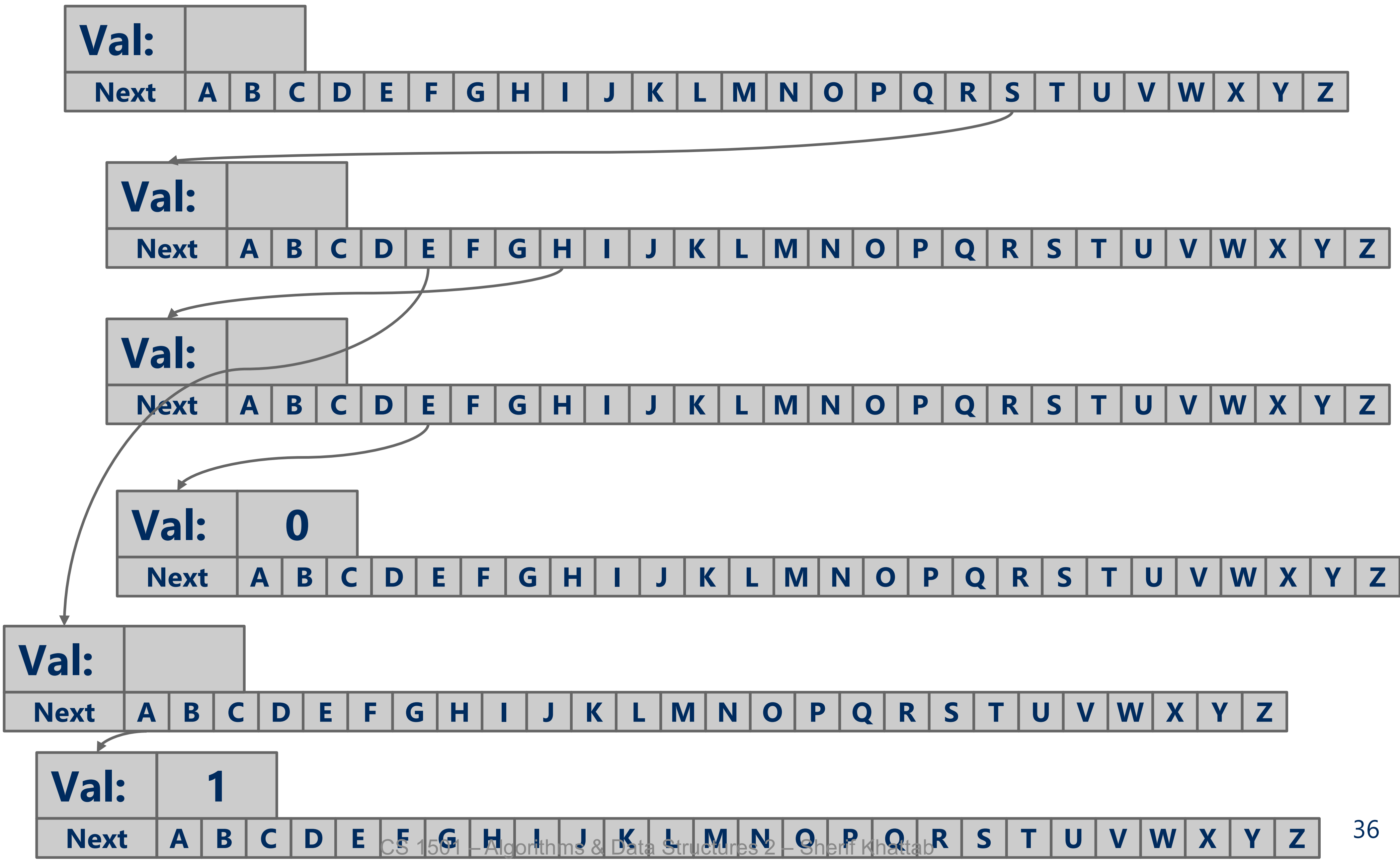
- See TrieSt.java
    - Implements an R-way trie
- Basic node object:

Where R is the branching factor

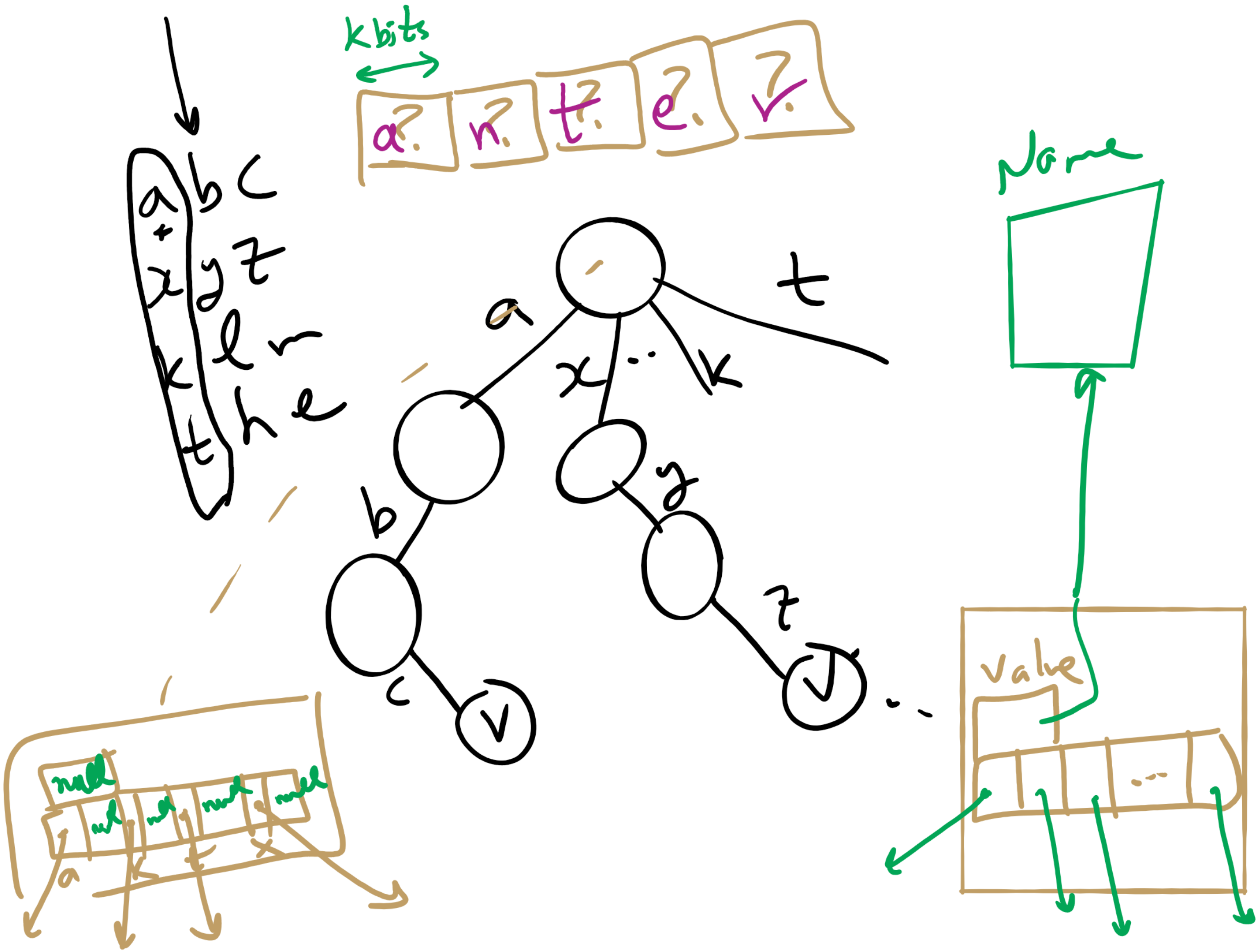
```
private static class Node {
    private Object val;
    private Node[] next = new Node[R];
}
```

- Non-null **val** means we have traversed to a valid key
- Again, note that keys are not directly stored in the trie at all

R-way trie example



R-way RST

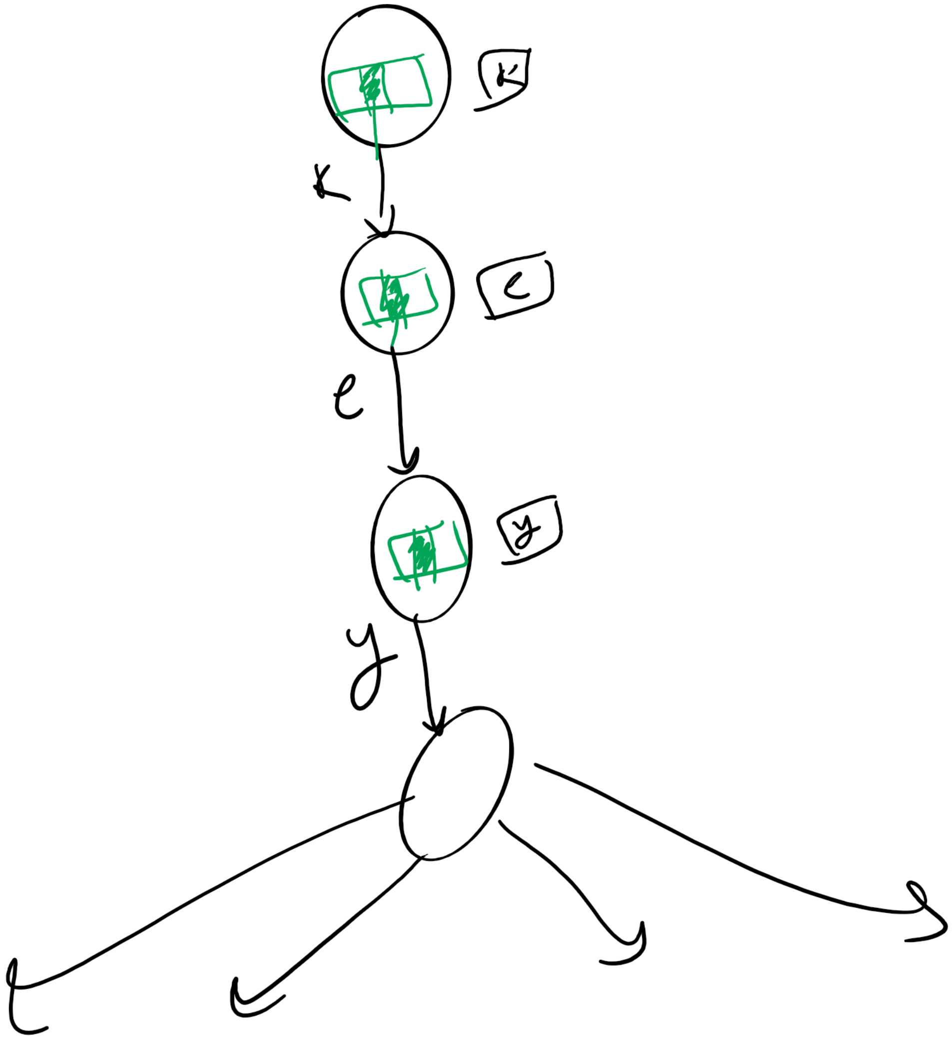


Summary of running time

	insert	search hit	search miss
binary RST	$\Theta(b)$	$\Theta(b)$	$\Theta(\log_2 n)$ on average
multi-way RST	$\Theta(w)$	$\Theta(w)$	$\Theta(\log_R n)$

So what's the catch with R-way RST?

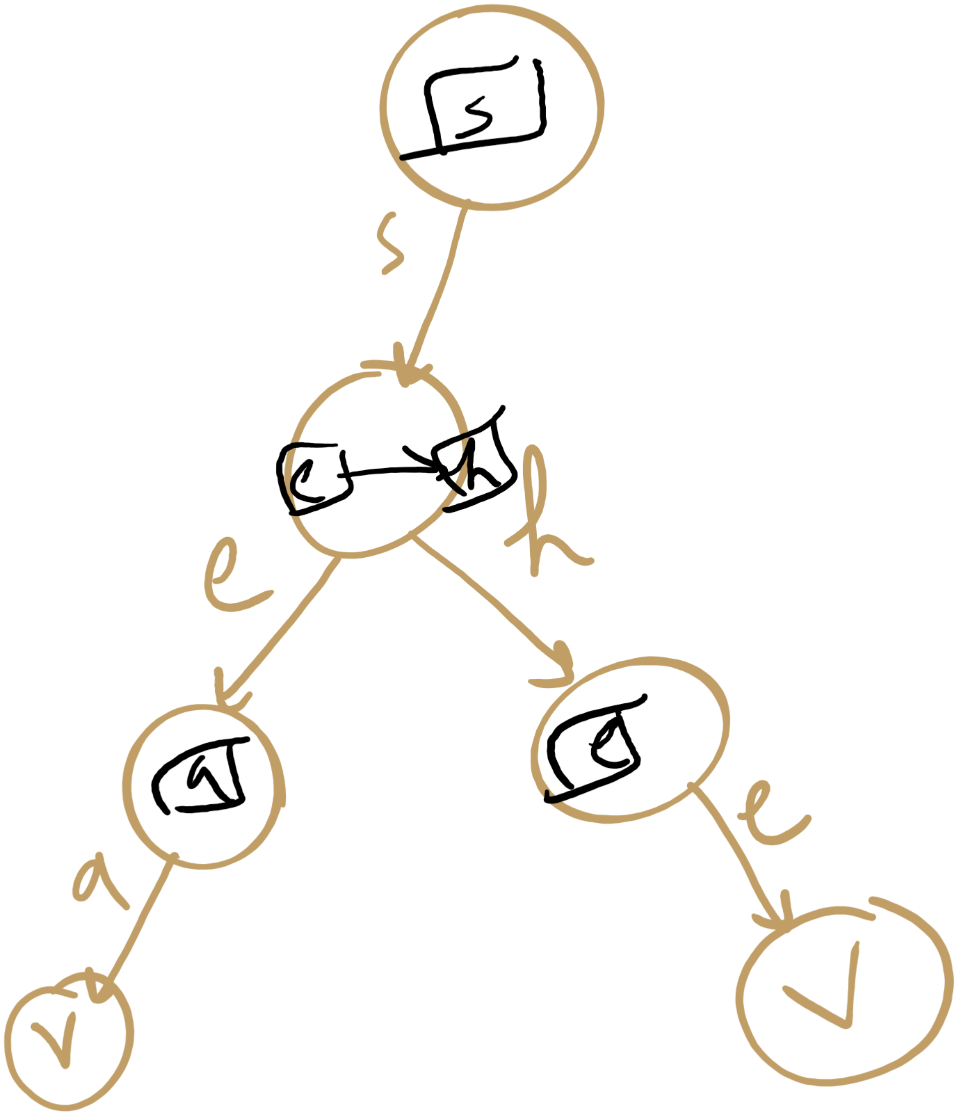
- Space!
  - Considering 8-bit ASCII, each node contains 2^8 references!
  - This is especially problematic as in many cases, alot of this space is wasted
    - Common paths or prefixes for example, e.g., if all keys begin with "key", thats 255*3 wasted references!
    - At the lower levels of the trie, most keys have probably been separated out and reference lists will be sparse



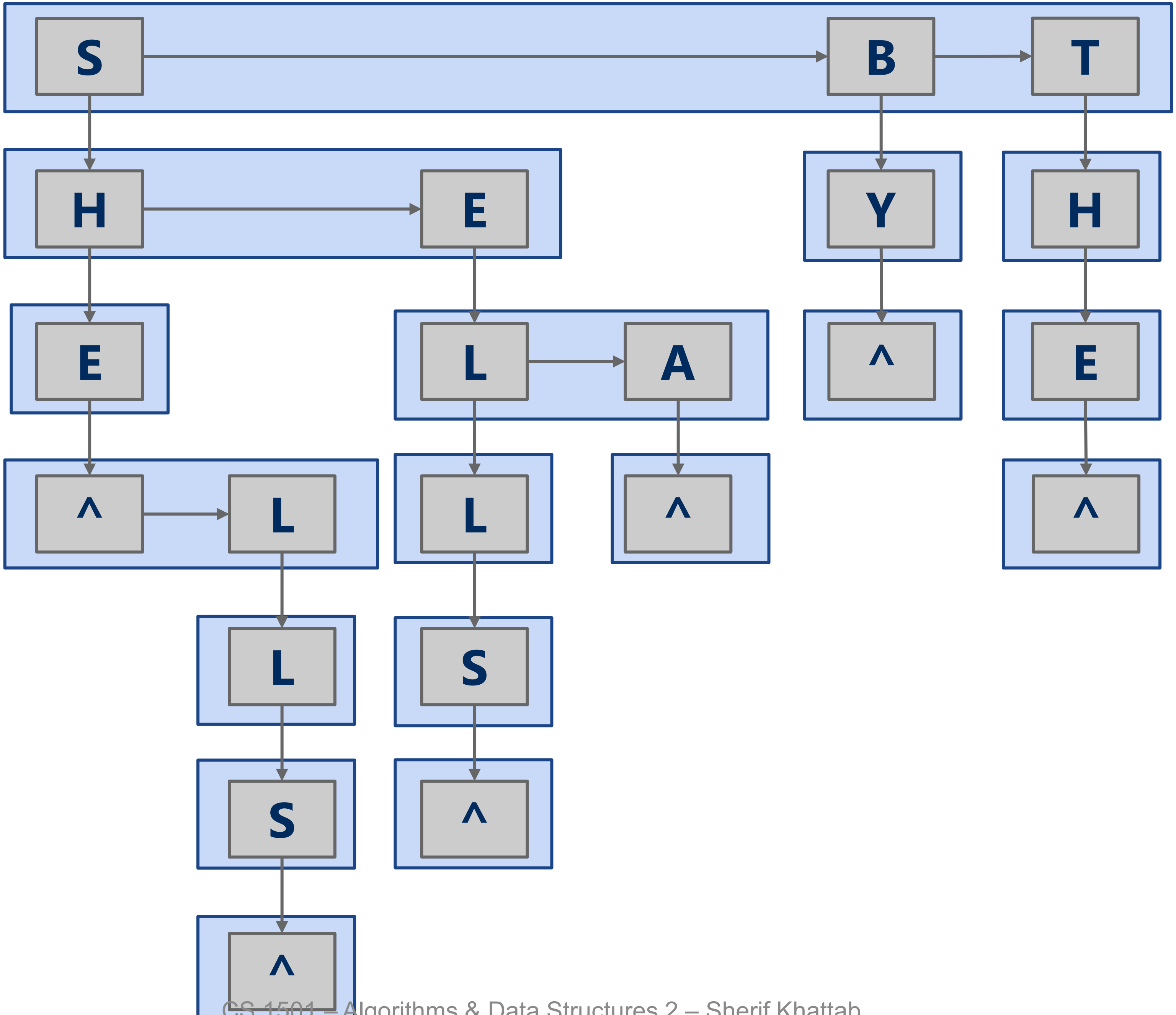
De La Briandais tries (DLBs)

Main idea: replace the .next array of the R-way trie with a linked-list

DLB Example



DLB Example 2: nodes vs. nodelets

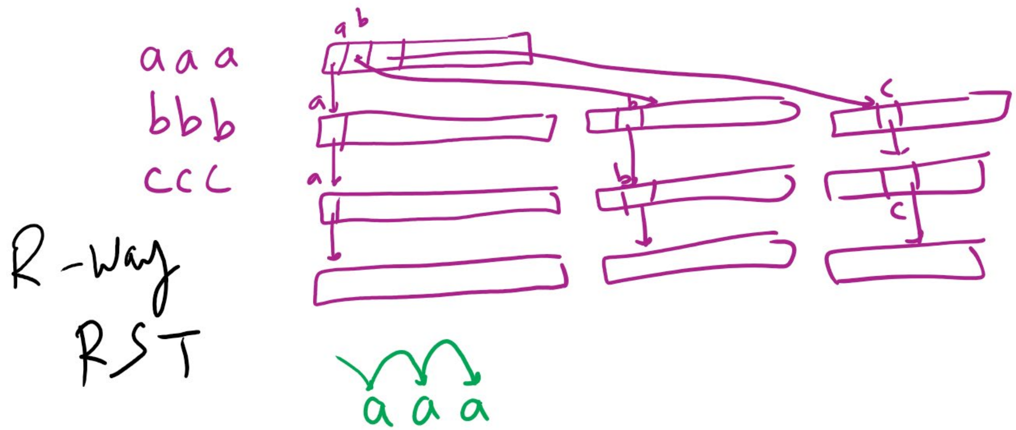


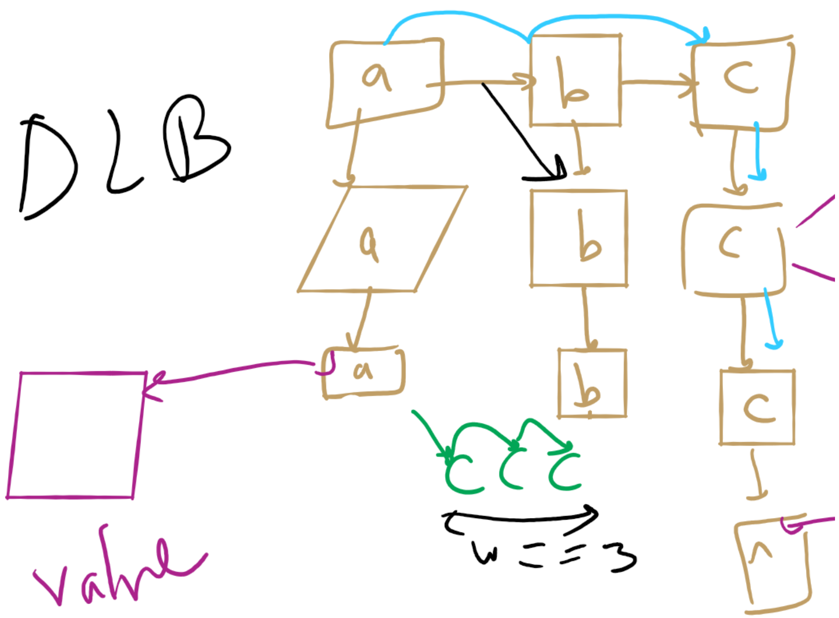
DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?

	search hit insert	
R-way RST	$\Theta(w)$	
DLB	$\Theta(wR)$	

R-way RST vs. DLB





$\Theta(w \cdot 1)$

#characters in the key

$\Theta(w \cdot R)$

Runtime Comparison for Search Trees/Tries

	Search hit	Search miss (average)	insert
BST	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
RB-BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
DST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
RST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
R-way RST	$\Theta(w)$	$\Theta(\log_R n)$	$\Theta(w)$
DLB	$\Theta(wR)$	$\Theta(\log_R n \cdot R)$	$\Theta(w \cdot R)$

Final notes on Search Tree/Tries

- We did not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
  - Ternary search Tries
  - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text